Lecture 2

Instruction Sets, Pipelining

RISC Vs CISC

• CISC (complex instruction set computer)
  - VAX, Intel X86, IBM 360/370, etc.
• RISC (reduced instruction set computer)
  - MIPS, DEC Alpha, SUN Sparc, IBM 801
RISC vs. CISC

• Characteristics of ISAs

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable length</td>
<td>Single word instruction</td>
</tr>
<tr>
<td>instruction</td>
<td>instruction</td>
</tr>
<tr>
<td>Variable format</td>
<td>Fixed-field decoding</td>
</tr>
<tr>
<td>Memory operands</td>
<td>Load/store architecture</td>
</tr>
<tr>
<td>Complex operations</td>
<td>Simple operations</td>
</tr>
</tbody>
</table>

The historical background:
- In first 25 years (1945-70) performance came from both technology and design.
- Design considerations:
  - small and slow memories: compact programs are fast.
  - small no. of registers: memory operands.
  - attempts to bridge the semantic gap: model high level language features in instructions.
  - no need for portability: same vendor application, OS and hardware.
  - backward compatibility: every new ISA must carry the good and bad of all past ones.

Result: powerful and complex instructions that are rarely used.
Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td></td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

Simple instructions dominate instruction frequency

RISC vs. CISC Instruction Set Design

- Emergence of RISC
  - Very large scale integration (processor on a chip)
  - Registers - load/store ISA. Micro-store occupied about 70% of chip area: replace micro-store with registers.
  - Increased difference between CPU and memory speeds.
  - Complex instructions were not used by new compilers.
  - reduced reliance on assembly programming, new ISA can be introduced.
  - standardized vendor independent OS (Unix) became very popular in some market segments (academia and research) - need for portability

- Early RISC projects: IBM 801 (America), Berkeley SPUR, RISC I and RISC II and Stanford MIPS.
The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type
  - I-type
  - J-type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td>31 26 21 16 11 6 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>31 26 21 16 11 6 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the "op" field
  - address/ immediate: address offset or immediate value
  - target address: target address of the jump instruction

MIPS Instruction Layout

- I-type instruction
  - OPCODE 5 5 16
  - Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt = rs op immediate)
  - Conditional branch instructions (rs is register, rd unused)
  - Jump register, jump and link register
    (rd = 0, rs = destination, immediate = 0)

- R-type instruction
  - OPCODE 5 5 5 5 5 6
  - Register-register ALU operations: rd – rs funct rt
    - Function encodes the data path operation: Add, Sub, ...
    - Pseudoinstructions special registers and moves

- J-type instruction
  - OPCODE 26
  - Jump and jump and link
    - Trap and return from exception

© 2003 Elsevier Science (USA). All rights reserved.
**MIPS Addressing Modes/Instruction Formats**

- All instructions **32** bits wide

**Register (direct)**

```
  op  |  rs |  rt |  rd |
```

**Immediate**

```
  op  |  rs |  rt | immed |
```

**Displacement**

```
  op  |  rs |  rt | immed |
```

**PC-relative**

```
  op  |  rs |  rt | immed |
```

Summary: Instruction Set Design (MIPS)

- Use general purpose registers with a load-store architecture: **YES**
- Provide at least 16 general purpose registers plus separate floating-point registers: **31 GPR & 32 FPR**
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred: **YES**: 16 bits for immediate, displacement (disp=0 => register deferred)
- All addressing modes apply to all data transfer instructions: **YES**
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size: **Fixed**
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: **YES**
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: **YES**
- Aim for a minimalist instruction set: **YES**
Pipelining: 5-stage Execution

- 5 stage “RISC” load-store architecture

1. Instruction fetch (IF):
   - get instruction from memory/cache
2. Instruction decode, Register read (ID):
   - translate opcode into control signals and read regs
3. Execute (EX):
   - perform ALU operation, load/store address, branch outcomes
4. Memory (MEM):
   - access memory if load/store, everyone else idle
5. Writeback/retire (WB):
   - write results to register file

Solution

- Overlap execution of instructions
  - Start instruction on every cycle, e.g. the new instruction can be fetched while the previous one is decoded - pipeline. Each cycle performing a specific task; number of stages is called pipeline depth (5 here)
Pipeline Progress - Instn moves with all control signals, addresses, data items => different register lengths at different stages

Pipelined Control
- Group control lines by pipeline stage needed
- Extend pipeline registers with control bits
A pipeline with multi-cycle FP operations:

**Arithmetic Pipeline: Ex. MIPS R4000**
Pipeline Hazards

- Hazards are caused by conflicts between instructions. Will lead to incorrect behavior if not fixed.
  - Three types:
    - **Structural**: two instructions use same h/w in the same cycle - resource conflicts (e.g. one memory port, unpipelined divider etc).
    - **Data**: two instructions use same data storage (register/memory) - dependent instructions.
    - **Control**: one instruction affects which instruction is next - PC modifying instruction, changes control flow of program.

Handling Hazards

- **Force stalls or bubbles in the pipeline.**
  - Stop some younger instructions in the stage when hazard happens
  - Make younger instr. Wait for older ones to complete
  - Implementation: de-assert write-enable signals to pipeline registers

- **Flush pipeline**
  - Blow instructions out of the pipeline
  - Refetch new instructions later - solving control hazards
  - Implementation: assert clear signals on pipeline registers